Chapter 11 Evaluation

This dissertation proposed, in Chapter 3, specified, in Chapters 4 through 9 and in Appendix A, and investigated, as described in Chapter 10 and in Appendices B through E, an approach to provide automated assistance to designers of concurrent software for real-time applications. The current chapter provides an evaluation of the approach. First, a summary evaluation is presented, followed by a more detailed discussion of the approach as applied to the cases studies described in Appendices B through E. The detailed discussion considers two, main aspects: 1) the modeling and analysis of specifications and 2) the generation and representation of designs. The detailed discussion helps to identify aspects of the approach that proved difficult to automate. The evaluation closes by considering the quality of the designs generated by CODA.

11.1 Summary Evaluation

The approach, as proposed in this dissertation, to automate the generation of concurrent designs from data/control flow diagrams is evaluated in three respects. First, the approach is evaluated against the research objectives identified in (Chapter 3), section 3.3. Second, the approach is evaluated against other approaches, as outlined in (Chapter 3), section 3.1. Third, the strengths and weaknesses of the approach are delineated.

11.1.1 Evaluation Against Research Objectives

One research objective called for using heuristics from an existing design method, for designing both tasks and modules in a concurrent design. This objective is satisfied through the specification of expert-system rules created from heuristics included in the CODARTS design method.

A second research objective was to provide two-way traceability between the data/control flow diagram and the resulting design. This objective is satisfied by specifying a Traces To/From relationship in the design meta-model.

A third research objective was to provide a basis for checking the resulting design for completeness with respect to the specification and for consistency with respect to the design meta-model. The traceability relationship between a given data/control flow diagram and a resulting design allows an assessment as to whether or not all specification elements are allocated to one or more components in the design. The consistency between generated designs and the design meta-model is evaluated by checking assertions whenever a design report is written to disk.

A fourth research objective was to capture the rationale for each design decision. This objective is satisfied by maintaining a decision history for each entity in the design meta-model. The decision histories can be viewed interactively and are also saved to a text file for each task and module whenever a design report is written to disk.

A fifth research objective was to allow alternative designs to be generated from the same specification based upon variations in the intended target environment. This objective is satisfied by providing a representation for target environment descriptions and then by writing various design-decision rules to consider aspects of the target environment description when generating a design. For the research reported in this dissertation alternate designs can be generated based upon: 1) differences in task inversion threshold, 2) variations in availability of message queuing services, and 3) differences in the maximum number of inter-task signals allowed by a specific run-time system. The remaining information contained in the representation for target environment descriptions provides for generating alternate designs during a design configuration phase. The configuration of designs is left as future research.

A sixth research objective was to elicit information from a designer only when such information is essential and cannot be inferred. This objective is satisfied through specification of a set of concept-classification rules that derive semantic interpretations of data/control flow diagrams, and also through an information elicitation process. In some cases semantic inferences could not be determined definitively; in some cases semantic inferences could be made only after additional information is provided by a designer; in a few cases semantic inferences could not be made.

The seventh, and final, research objective was to vary decision-making responsibility depending upon the designer’s level of experience. This objective is satisfied by permitting designers to identify their level of experience and then by considering this self-declared, experience level when making design decisions that might benefit from a designer’s insight. Decision-making is varied in two ways. First, at times when an experienced designer is not available the design generator makes default decisions. Second, at times when an experienced designer is not available the designer generator chooses not to consider certain design decisions requiring subtle reasoning.

11.1.2 Comparison Against Other Approaches

An earlier chapter, Chapter 3, section 3.1, evaluated a number of research efforts that attempt to automate the generation of designs from flow graphs. How does the approach proposed and specified in this dissertation compare against these earlier research efforts? Table 10 adds a column to a table, Table 1, presented in Chapter 3. The additional column enumerates the traits of the prototype, COconcurrent Designer's Assistant (CODA), described in Chapter 10. This table provides a convenient means of comparing CODA against these other approaches.

CODA is the only approach that represents data/control flow diagrams as the input model. CODA provides human-readable task and module specifications as output, in addition to an internal representation of the design meta-model defined in Chapter 5 of this dissertation. The other approaches produce either some form of structure chart or an input into a design simulator. CODA uses the COBRA method to interpret the data/control flow diagram and the CODARTS method to generate concurrent designs. The other approaches use either criteria from Structured Design or unique mapping rules,

Table 10. Summary Comparison of Various Approaches to Design Generation

	Research Effort				
Trait	CAPO	EARA	STES	SARA	CODA
Input Model	DFDs	DFDs	DFDs	DFDs & SVDs	D/CFDs
Output Model	Structure Charts	Structure Charts	Structure Charts	GMB Objects & SARA Structural Model	Task & Module Specifications, Design Meta-Model
Decision Method	Coupling & Cohesion	Structured Design	Structured Design	Mapping Rules	COBRA & CODARTS
Underlying Techniques	Clustering Algorithms	Formal Rule Rewriting	Expert System Rules	Expert System Rules	Expert System Rules, Semantic Data Modeling
Completeness & Consistency Checking	No	No	No	No	Yes
Traceability	Implicit	Explicit	Implicit	Implicit	Explicit
Design Rationale Capture	No	No	No	No	Yes
Requires Elicitation From Designer	No	Yes	Yes	Yes	Yes
Varies Elicitation With Designer’s Experience	No	No	No	No	Yes
Varies Design With Target Environment	No	No	No	No	To a limited extent.
Connects With CASE Tool	No	No	Yes	Yes	No

not based on an existing design method. CODA uses expert-system rules, but also applies semantic data modeling to both the input specification and the evolving design. CODA treats instances of both the specification and the design as an object-oriented database implementation of the relevant semantic data models, the specification and design meta-models, respectively. This treatment permits designs to be generated, as with the other approaches, but also allows designs and specifications to be queried interactively.

CODA is the only one of the approaches listed in Table 10 that: 1) provides explicit checking for completeness of the generated design against the data/control flow diagram and for consistency of the generated design against the design meta-model, 2) captures the design rationale, and 3) varies the design based on variations in the intended target environment. CODA is one of only two approaches that provides explicit traceability, and CODA is the only prototype that can provide traceability in two directions, that is, from specification element to design element and from design element to specification element. Among the approaches that require elicitation from the designer, CODA is the only approach that varies the nature of the elicitation based upon a designer’s declared level of experience. Although CODA does not include connection to a CASE tool, the input to CODA is a text file that could be generated from existing CASE databases using an extraction and formatting program.

11.1.3 Strengths and Weaknesses of the Proposed Approach

Though the approach embodied in CODA meets the research objectives established for this dissertation, CODA exhibits some definite strengths and weaknesses. CODA's main strengths include the following.

- CODA can make a substantial number of semantic interpretations from an RTSA data/control flow diagram without consulting a designer.
- CODA can check automatically a data/control flow diagram for proper semantic classification.
- CODA elicits additional information from a designer only where necessary.
- CODA generates designs based upon heuristics contained within a preexisting method for producing concurrent designs.
- CODA refers subtle design decisions to an experienced designer. When an experienced designer is unavailable, CODA still produces a design that is complete with respect to the data/control flow diagram.
- CODA can check automatically the completeness of each generated design against the specification and the consistency of each generated design against the design meta-model.
- CODA can view both specification and design as interconnected object-oriented databases that can be queried about their structure and about relationships within and between them.

- CODA captures design histories, including design rationale, for each design element, and these histories can be viewed interactively or saved for off-line review.
- CODA can vary, to a limited extent, designs based upon relevant characteristics of the intended target environment.

CODA's main weaknesses include the following.

- When synchronous functions might be allocated among several existing tasks or modules, CODA does not possess the application-specific knowledge required to infer the best allocation, but must, instead, consult a designer.
- CODA must elicit specification addenda from the designer to learn about potentially important factors that cannot be represented on, nor inferred from, a data/control flow diagram.
- In many instances CODA cannot infer the synchronization requirements for data flows sent between tasks but must, instead, elicit this information from an experienced designer or must make a default decision.
- In some instances, such as module operations, parameter definitions, and task-module calling sequences, CODA encodes specific, mapping decisions, where as a human designer might wish to consider alternate approaches.

11.2 Analysis of the Case Studies

The preceding sections provided a summary evaluation of the approach. A more detail discussion of the approach, as applied to the case studies described in Appendices B through E, follows. This detailed discussion contains some analysis to support the preceding summary evaluation.

11.2.1 Modeling and Analysis of Specifications

To automate the generation of concurrent designs, a means is required to model and analyze specifications, that is, data/control flow diagrams. As described in Chapter 4 and Appendix A, this problem is solved by constructing a semantic meta-model that consists of a concept hierarchy, including appropriate axioms for each concept, and a set of classification rules. How well does this solution succeed?

11.2.1.1 Semantic Interpretation of Flow Diagrams

Table 11 presents an overview of the classification process across all specifications, as provided in the four case studies described in Appendices B through E. Each element of a specification can be depicted on a data/control flow diagram using one of seven RTSA syntactic components. For conciseness, these syntactic elements are represented in five rows within Table 11. Row two, Transformations, includes both data and control transformations; row seven, Data Flows, includes both unidirectional and bi-directional data flows. Row one of Table 11 represents all nodes on a data/control flow diagram, row five represents all arcs, and row 8 represents all elements.

The first column in Table 11 simply gives the total number of each type of element that appears on the data/control flow diagrams in the case studies; for example, the case-study diagrams contained 358 elements as follows: 1) 125 nodes of which 79 were transformations, 28 were terminators, and 18 were data stores and 2) 233 arcs of which 91 were event flows and 142 were data flows.

The remaining five columns in Table 11 represent the manner in which classification is achieved for the elements represented by each intersecting row. Five possibilities exist as shown in the legend following the table.

Table 11. Classifications Over All Specifications for the Case Studies

	Total	@	#	=	?	+
All Nodes	125	28 (22%)	18 (14%)	59 (47%)	8 (6%)	12 (10%)
Transformations	79			59 (75%)	8 (10%)	12 (15%)
Terminators	28	28				
Data Stores	18		18			
All Arcs	233			231 (99%)		2 (1%)
Event Flows	91			91		
Data Flows	142			140 (99%)		2 (1%)
All Elements	358	28 (8%)	18 (5%)	290 (81%)	8 (2%)	14 (4%)

<u>Column Headings</u>

@	Classified By The Designer
#	Directly Representable Within The Specification Meta-Model
=	Classified By CODA
?	Tentatively Classified By CODA, Confirmed or Overridden By The Designer
+	Classified By CODA, After Eliciting Additional Information From The Designer

In certain cases a specification element can only be classified by the designer (column headed by **@** symbol). As shown in Table 11 this occurs for each and every terminator in the case studies. No means exists to infer automatically whether a given terminator represents a device, a user role, or an external subsystem. In the case of data stores, a syntactic element from RTSA represents directly the same concept within the specification meta-model (column headed by **#** symbol). The remaining columns represent cases where CODA can infer the existence of a semantic concept by analyzing the context in which syntactic elements appear on a diagram. Three possibilities exist. First, CODA can identify a semantic concept without any additional information (column headed by **=** symbol). Second, CODA can make a tentative classification of a semantic concept, but must consult with the designer because the classification might be incorrect (column headed by **?** symbol). Third, CODA can make a classification only after the designer supplies some information that is not represented on the data/control flow diagram but that might exist in some external specification or in the designer's head (column headed by **+** symbol). For an ideal automated classifier, each element across all specifications would fall within one of two columns: **#**, represented directly or **=**, classified automatically.

A review of the last row of Table 11 indicates the degree of success achieved when CODA's automated classifier, using the classification rules specified in Appendix A, is employed against the data/control flow diagrams shown in Appendices B through E. For about 86% of the elements in the data/control flow diagrams the automated classifier succeeded without help. The designer specified a classification in those 8% of cases that represent terminators; these cases are not worth considering in detail. Cases that are worth considering in detail involve those 2% where CODA's classification must be confirmed by the designer or those 4% where CODA's classification can be made only after the designer supplies additional information. In these cases, shown in the columns, headed by **?** and **+**, respectively, of Table 11, a large difference exists between the classification of arcs, where 99% are classified without help, and the classification of transformations, where only 75% are classified without help.

11.2.1.1.1 Problems Classifying Data Flows

Only two, data flows, from among the 233 arcs considered in the case studies, cannot be classified without consulting the designer. Both of these appear in the data/control flow diagram for the elevator control system (see Appendix D); in fact, the two data flows are related. A function, Scheduler, sends a data flow, Scheduler Request, to another function, Accept New Request, which also sends a data flow, Elevator Commitment, to the function Scheduler. CODA's automated classifier cannot determine which of these two data flows, if either, is sent in response to the other; therefore, the designer must be consulted. The designer should know or be able to determine that one or neither of the data flows is a response to the other. Of course, the designer might not know this information, so the automated classifier is prepared to make a default decision. Only in situations such as this one will CODA's automated classifier need to consult the designer to classify arcs on a data/control flow diagram. Thus, the performance of the automated classifier against arcs will depend on the number of these cases that exist in a given data/control flow diagram. The performance of CODA's automated classifier against functions appears less effective.

11.2.1.1.2 Problems Classifying Functions

For the case studies considered in this dissertation, only 75% of transformations could be classified automatically, while 10% could be classified tentatively, and 15% required additional information from the designer before a classification could be made. This information is reflected in Table 12, which also provides a breakdown of the classifications by case study.

Table 12. Classification of Transformations by Case Study

Case Study	**Transformations**	**=**	**?**	**+**
All Specifications	79	59 (75%)	8 (10%)	12 (15%)
Automobile Cruise Control	33	27 (82%)	6 (18%)	
Robot Controller	18	14 (78%)	1 (5%)	3 (17%)
Elevator Control	14	13 (93%)		1 (7%)
Remote Temperautre Sensor	14	5 (36%)	1 (7%)	8 (57%)

Table 12 illustrates that the performance of the classification rules is much better in the three case studies where the data/control flow diagram is constructed with the COBRA semantic model in mind. Considering only these three cases, the automobile cruise control and monitoring system (see Appendix B), the robot controller (see

Appendix C), and the elevator controller (see Appendix D), 83% (54/65) of transformations were classified automatically, 11% (7/65) could be classified tentatively, while only 6% (4/65) required additional information in order to make a classification. For the remote temperature sensor case study (see Appendix E), where the data/control flow diagram is developed using functional decomposition, the performance of CODA's automated classifier is singularly poor. The numerous aperiodic functions that appear in the data/control flow diagram provided by Nielsen and Shumate defy classification without assistance from the designer. A detailed analysis of specific problems follows.

11.2.1.1.2.1 Tentative Classifications

Eight transformations, encompassing about 10% of the transformations considered across the cases studies, could only be classified tentatively using CODA's automated classifier. As shown in Table 12, these include six transformations in the automobile cruise control and monitoring system and one transformation in each of two other cases studies, the robot controller and the remote temperature sensor. Each of these transformations involves the same type of situation. Whenever the classifier finds a function on a data/control flow diagram such that the function sends data only to data stores and/or passive, device-interface objects, then, if the function is not classified otherwise, the CODA tentatively identifies the function as synchronous. This tentative classification is based on the idea that in real-time systems updating data stores and writing to passive devices is generally a fast operation that should be completed atomically. The classification is not always correct, however, because an operation might take long enough that the designer chooses to view the function as asynchronous. The designer confirms the tentative classification in seven of the eight cases among the data/control flow diagrams given in the appendices. In the remote temperature sensor case study, the designer overrides CODA's tentative classification where a function, Maintain Temperature Table, updates a data store, Temperature Table. The designer overrides CODA based on his application-specific knowledge the data store is large enough and/or the algorithm is time-consuming enough to warrant asynchronous processing. This information is not available to CODA but might be available to the designer. When the designer does not know whether to override the CODA's tentative decision, then the decision stands.

11.2.1.1.2.2 Assisted Classifications

In some situations CODA's classifier recognizes that additional information might be available that can help make a better classification. In these situations, represented in the last column of Table 12, CODA consults the designer to see what other information exists. Lacking additional information, CODA makes a default classification. Table 12 indicates twelve instances among the case studies where the designer is consulted to help classify a transformation. These twelve instances represent two general situations. The first situation occurs when a function is triggered by a control transformation, yet the triggered function receives input data from some other transformation. In situations of this type, the classifier recognizes that the triggered function might not be able to execute during the triggered state-transition. The designer is consulted on this question. CODA then makes the best classification based upon any additional insight gleaned from consulting the designer. The second, and more frequent, situation occurs when a function receives input from only a single source, or the same input from multiple sources. In such situations, CODA recognizes that the function might be classified as either a synchronous or asynchronous function depending upon certain factors. In order to gauge these factors, CODA consults with the designer. The designer is asked for one or two facts depending upon the CODA's needs. First, the designer is asked if the time taken to execute a function might unduly delay its invoking function(s). The designer might also be asked whether the algorithm embodied within the function takes substantial time to execute.

11.2.1.2 Extensions and Restrictions to RTSA

The foregoing cases where the designer is consulted represent situations where information cannot be inferred directly from a data/control flow diagram. To limit such situations to the minimum number possible, the specification meta-model represents a restricted application of RTSA notation, as embodied primarily in the ideas underlying COBRA. In addition, the specification meta-model admits a few extensions and requires a few other restrictions to RTSA notation. These extensions and additional restrictions, beyond COBRA, are worth describing.

Two extensions to RTSA notation are included in the specification meta-model, as described in Chapter 4 and Appendix A. First, an event flow and data flow can be emitted in parallel from a terminator to a transformation. Each such event flow represents the arrival of an interrupt from the terminator. (The data flow in this case represents input data.) Second, a terminator named "System" is assumed to exist in every specification. Event flows from this special terminator are used to represent timer events. These two extensions make it possible to represent timers and interrupts on a data/control flow diagram, and also help to automatically classify various transformations.

One major restriction imposed on RTSA notation by the specification meta-model is the omission of continuous data flows. This restriction is not very significant for digital computer systems because most real-time software systems use discrete events and data. A number of minor restrictions are imposed by the axioms defined for each concept in the specification meta-model. None of these restrictions limits severely the range of data/control flow diagrams that can be represented. Perhaps the most significant restriction is that parallel data flows in the same direction are not permitted between any pair of nodes. To compensate for this restriction, complex data flows must be described using a data dictionary. The restrictions imposed by the axioms serve primarily to facilitate automated checking of the syntactical and semantic correctness of data/control flow diagrams.

11.2.1.3 Eliciting Helpful Information

As demonstrated in a preceding discussion, a software designer possesses access to information beyond the data/control flow diagrams for a software system. Such information can exist in a textual specification, in pseudo-code, and in state-transition diagrams. Some of this amplifying information, while not needed to classify concepts on a data/control diagram flow, is critical to the generation of concurrent designs and must be provided by a designer. Other information can improve the quality of CODA's decision-making and, thus, a designer is asked, where possible, to provide this additional information. In several of the case studies CODA elicits such helpful information.

11.2.1.3.1 Eliciting Node Cardinality

In two case studies, multiple instances are required for transformations, data stores, or terminators. The robot controller problem requires six axis controllers, though only one is shown on the data/control flow diagram. In the elevator control system, several transformations can require numerous instances, depending on the configuration of the building to be served. CODA elicits these cardinalities from the designer.

11.2.1.3.2 Eliciting Locked-State Events

CODA also elicits locked-state events. Only one locked-state event, Ended, exists in the robot controller case study, described in Appendix C. Numerous locked-state events appear in the elevator control system case study, presented in Appendix D. The locked-state events clearly include: Door Closed, Elevator Started, Elevator Stopped, Destination Up, Destination Down, and No Destination. Consideration of three other events, however, present more difficulty. Two events, Up Request and Down Request, arrive from an asynchronous function, Accept New Request, and can occur at any time whether the elevator controller is expecting them or not. These events fail one criterion for a locked-state event. The third event, Approaching Requested Floor, arrives from an asynchronous function, Check This Floor. This event cannot, however, arrive unless the elevator controller is expecting it because the sensor that causes the input that generates the event cannot detect the input unless the elevator controller has started the elevator moving. Approaching Requested Floor, then, is a locked-state event.

11.2.1.3.3 Eliciting Exclusion Groups

In the absence of other mechanisms, CODA asks the designer to specify any mutually exclusive execution: 1) among each set of enabled functions that are managed by the same control transformation and 2) among the set of state-independent periodic and asynchronous functions in a given data/control flow diagram. Mutual exclusion among enabled functions appears in the automated cruise control system, described in Appendix B. Three enabled functions, Resume Cruising, Increase Speed, and Maintain Speed, managed by the transformation, are never enabled in the same state. The designer places these transformations into an exclusion group.

Mutual exclusion among state-independent functions appears in the remote temperature sensor case study, where one periodic function, Send Old DP, and one asynchronous function, Get New DP, cannot execute simultaneously. This can be determined by reading the textual specification for the remote temperature sensor. The system uses a stop-and-wait protocol; only one packet is ever outstanding at a given time. This means that the system is either waiting to retransmit an old packet or is free to send a new packet, but the system cannot be in both states at the same moment. The designer places these transformations into an exclusion group. In addition, an asynchronous function, Maintain Temperature Table, and a periodic function, Monitor Periodic Query, cannot both access a shared data store, Temperature Table, simultaneously; thus, the designer places these two transformations into an exclusion group.

11.2.1.3.4 Eliciting Aggregation Groups

The need to represent aggregation relationships appears in the elevator control system case study, described in Appendix D. Each elevator is composed of a set of elements: a motor, a door, buttons, lamps, and a controller. The designer places these elements in an aggregation group.

11.2.2 Generation and Representation of Designs

Only after a specification is analyzed and all required information is available can design generation commence. Design generation is based upon heuristics from the CODARTS design method, which are encoded as expert-system rules. The following sections consider the degree of difficulty faced when encoding various heuristics. In addition, cases are identified where an experienced designer can provide insight unavailable to the design generator. These topics are addressed for each of the CODARTS design phases: Task Structuring, Task Interface Definition, Module Structuring, and Task and Module Integration.

11.2.2.1 Task Structuring

The structuring of tasks involves four decision-making processes. Each of these processes is considered in turn below.

11.2.2.1.1 Candidate Tasks

During the generation of concurrent designs for the case studies described in Appendices B through E, candidate tasks were identified for seven, distinct designs, as listed below.

- Automobile Cruise Control & Monitoring System, with Default TED (as specified in Chapter 5), and an experienced designer (C1)
- Automobile Cruise Control & Monitoring System, with Default TED, and a novice designer, so CODA applies its default decision-making (C2)
- Robot Control System, with Default TED, and an experienced designer (R1)
- Robot Control System, slightly altered specification, with Default TED, and an experienced designer (R3)
- Elevator Control System, Small Building, with Default TED, and an experienced designer (E1)
- Elevator Control System, Large Building, with Default TED, and an experienced designer (E4)
- Remote Temperature Sensor application, with TED simulating an Ada environment, and an experienced designer (RT)

Letter and number combinations, as indicated above in parentheses, are used to represent each of these designs in column headings for several tables that follow.

Table 13 displays the number of rule executions required to identify candidate tasks for each of the designs listed above. The table contains 12 rows, one for each of

Table 13. Rule Executions to Identify Candidate Tasks

CODARTS Criterion & Rule Name	C1	C2	R1	R3	E1	E4	RT	Total
I/O Task Structuring Criteria								
Asynchronous Device I/O Task: Asynchronous Device Interface	2	2	3	3	3	3	3	19
Periodic I/O Task: Periodic Device Interface	6	6	2	2				16
External Subsystem								
Internal Task Structuring Criteria								
Asynchronous Task: Asynchronous Algorithm			2	2	3	3	5	15
Asynchronous Task: Enabled Asynchronous Algorithm								
Asynchronous Task: Triggered Asynchronous Algorithm					1	1		2
Control Task: State-based Control	2	2	1	1	1	1		8
Periodic Task: Periodic Algorithm	6	6					2	14
Periodic Task: Enabled Periodic Algorithm	3	3						6
Periodic Task: Triggered Periodic Algorithm								
User Role Task: User Role								
Total	19	19	8	8	8	8	10	80

the eleven rules specified for candidate-task identification and one for the total rule executions for each design. The rows are organized into two groups, based upon categories of task structuring criteria included in CODARTS, I/O Task Structuring Criteria and Internal Task Structuring Criteria. Each rule is identified by the specific CODARTS task structuring criterion addressed by the rule, where applicable, followed by the name of the rule, as specified in preceding chapters of this dissertation. The table also contains eight columns, one for each of the seven designs and one for the total number of executions of each rule.

The heuristics required to identify candidate tasks were easily automated because previous analysis of the data/control flow diagram had produced a semantic interpretation for the various transformations. As shown in Table 13, all but four of the 11 rules were executed at least once. These four rules remain unexecuted because no transformation of the type recognized by each of these rule appears in any of the case studies.

11.2.2.1.2 Remaining Transformations

The next decision-making process allocates the remaining transformations, that is, those not leading to candidate tasks, among the set of candidate tasks identified by the previous decision-making process. Table 14 tabulates the rule executions required to make these decisions for the case studies. All but one of the rules listed in Table 14 executes without consulting the designer. One rule, Designer Specifies Allocation, consults the designer whenever a transformation representing a synchronous function connects directly to more that one candidate task. In such cases, allocation of the transformation might be made to any of the connecting tasks or to a separate task. The decision depends typically on a designer’s judgment about the specific application. For this reason, CODA consults the designer. When the designer cannot provide additional insight then CODA makes a default decision to allocate the transformation in question to

Table 14. Rule Executions to Allocate Remaining Transformations to Tasks

CODARTS Task Cohesion Criterion & Rule Name	C1	C2	R1	R3	E1	E4	RT	Total
Control Cohesion: Triggered Synchronous Function	4	4	3	3	1	1		16
Sequential Cohesion: Synchronous Function With Synchronous Outputs	6	6	2	2				16
Sequential Cohesion: Synchronous Function With Synchronous Inputs							2	2
Sequential Cohesion: Outputs Only Locked-State Event To Control Object								
Sequential Cohesion: Responding Synchronous Function								
Sequential Cohesion: Passive Device Interface Object	10	10			2	2		24
Sequential Cohesion: Aggregated Passive Device					3	3		6
Designer Choice: Implement Designer Allocation			2	1			3	6
Designer Choice: Designer Specifies Allocation			2	1			3	6
Total	20	20	9	7	6	6	8	76

a separate task. Another rule, Implement Designer Allocation, implements any choice made by the designer. For the case studies reported in this dissertation, CODA consulted the designer when allocating transformations to tasks during six of 70 rule executions, or in about 9% of the cases, excluding the implementation of the designer's choice. When viewed across the rule executions shown in Tables 13 and 14, CODA consulted the designer in only six of 150 rule executions, or in about 4% of the cases.

Two rules listed in Table 14 were not executed while generating designs for the case studies reported in this dissertation. These rules recognize situations that do not occur in any of the case studies. One rule, Outputs Only Locked-State Event To Control Object, recognizes synchronous functions that emit only locked-state events to a control object. For example, in the elevator control system specified in Appendix D, a transformation, Check This Floor, emits only a locked-state event, Approaching Requested Floor, to a control object, Elevator Controller. Should the transformation Check This Floor represent a synchronous function then the rule, Outputs Only Locked-State Event To Control Object, would apply. In the case study, however, the transformation, Check This Floor, represents an asynchronous function.

The other unexecuted rule, Responding Synchronous Function, listed in Table 14 recognizes situations where a synchronous function receives a data flow and then generates a data flow in response, but where the function generates no data flow on its own initiative. No such situation appears in the case studies.

11.2.2.1.3 Task Mergers

During the next decision-making process required to structure tasks, CODA considers whether any of the candidate tasks might be merged based upon various forms of cohesion criteria, as specified in CODARTS. In practice, consideration of task mergers can be made either before or after remaining transformations are allocated to tasks. In fact, during the research associated with this dissertation, both orderings were tested and found to produce the same results.

The automation of heuristics for considering task mergers proved to be somewhat difficult. One difficulty arises because multiple situations can exist in a specification that lead to different decisions about which tasks to merge. In addition, some forms of task mergers will still allow other cohesion criteria to be applied to the merged tasks, while other forms of task merger bar the application of additional cohesion criteria. For example, when a periodic input task is merged with a control task then the merged task is not a candidate for combining with other periodic tasks based on temporal cohesion. To address these situations, careful consideration must be given to preferred rule orderings when multiple forms of cohesion might be applied in the same situation. A second difficulty appears when considering the application of temporal cohesion to tasks without identical periods. The periods of such tasks must share a common factor but, to avoid combining tasks of differing priority, the periods must be relatively close to each other and the tasks with the closest qualified periods should be considered first. Writing a rule to recognize this combination of conditions proved difficult. Even when such a rule exists, the actual decision to combine tasks under these conditions requires application-specific knowledge unavailable to CODA. For these reasons, temporal cohesion is not applied to tasks without identical periods, unless an experienced designer is available for CODA to consult. When an experienced designer is available, CODA recognizes situations where tasks without identical periods might be combined and then refers those situations to the designer for a final decision. When an experienced designer is unavailable then CODA will not combine tasks without identical periods.

Table 15 enumerates the rule executions required to consider task mergers for each of the case studies described in Appendices B through E. Each specified rule executes at least once, except for one rule, Periodic Tasks With Multiple Instances. The unexecuted rule recognizes tasks with multiple, identical instances. Such tasks can be inverted based upon the temporal and functional cohesion criteria of CODARTS; however, no situation of this type appears among the case studies.

11.2.2.1.4 Resource Monitors

During the final decision-making process required to structure tasks, CODA considers the need for resource monitor tasks in the design. Resource monitor tasks are required whenever two or more tasks generate output for a passive device and when the outputs for each task must be generated atomically and in order. Two rules recognize such situations. One rule recognizes when multiple tasks write to the same passive device. The second rule recognizes when a single, multiple-instance task writes to a passive device. Table 16 lists the applicable rule executions for the case studies.

Table 15. Rule Executions to Consider Task Mergers

CODARTS Cohesion Criterion & Rule Name	C1	C2	R1	R3	E1	E4	RT	Totals
Mutual Exclusion: Controlled Task Exclusive Execution	2	2						4
Mutual Exclusion: Independent Task Exclusive Execution							2	2
Task Inversion: Task Inversion					3	5		8
Sequential Cohesion: Exclusive Input To Control Task	1	1						2
Sequential Cohesion: State-Dependent Input To Control Task						2	2	4
Temporal/Functional Cohesion: Periodic Task With Multiple Instances								
Temporal/Functional Cohesion: Periodic Tasks With The Same Periods	5	5						10
Temporal/Functional Cohesion: Periodic Tasks With Harmonic Periods	1							1
Totals	9	8			5	7	2	31

Table 16. Rule Executions to Consider Resource Monitors

CODARTS Task Structuring Criterion & Rule Name	C1	C2	R1	R3	E1	E4	RT	Total
Resource Monitor Task: Multi-Task Access								
Resource Monitor Task: Multi-Instance Task Access					2			2
Total					2			2

Only the elevator control system case study requires resource monitors. A multiple-instance task, Elevator Controller, writes to two passive output devices, floor lamps and direction lamps. This situation disappears when the elevator control system is expanded to a larger building because CODA inverts the Elevator Controller tasks into a single task. This inversion occurs in an effort to reduce the task switching overhead.

Other situations might arise where resource monitor tasks prove useful. For example, if a single-instance task writes output to a passive device and the passive device takes substantial time to perform the processing associated with the device and the single-instance task has useful work to perform, then the passive device might be given a monitor task. No rule of this nature was defined for this dissertation. Should such a rule be defined then an experienced designer would need to be consulted at some stage in the process to determine the time required to perform output for each passive device.

11.2.2.2 Task Interfaces

The definition of task interfaces occurs as a series of five decision-making processes, discussed in turn below. First, however, three alternate designs must be introduced. In the case studies, a number of alternate designs are generated based upon differences in the intended target environment, or upon differences in inter-task message priorities. These alternate designs lead to variations in the interfaces between tasks; thus, the tables enumerating rule executions for the definition of task interfaces include three additional columns, as listed below.

- Robot Controller System, when TED indicates no message queuing available and the designer is experienced (R2)
- Elevator Control System, with Default TED, where an experienced designer assigns message priorities (E2)
- Elevator Control System, when TED indicates priority queues available, where an experienced designer assigns message priorities (E3)

11.2.2.2.1 External Task Interfaces

During the first decision-making process required to define task interfaces, CODA examines the arcs on a data/control flow diagram and determines which arcs map to task inputs, outputs, interrupts, and timers and which arcs map to data flows and event flows between tasks. These decisions were quite straightforward to automate. CODA requires no interaction with a designer to make any of these decisions. Table 17 shows the rule executions required for the case studies.

Table 17. Rule Executions to Allocate External Task Interfaces

Rule	C1	C2	R1	R2	R3	E1	E2	E3	E4	RT	Totals
Timer Event	15	15	2	2	2					2	38
Interrupt	2	2	3	3	3	3	3	3	3	3	28
Data Input	11	11	3	3	3	5	5	5	5	2	53
Data Output	3	3	3	3	3	5	5	5	5	2	37
Inter-Task Exchange	18	18	19	19	19	10	10	10	8	12	143
Totals	49	49	30	30	30	23	23	23	21	21	299

11.2.2.2.2 Control And Event Flows

Once the inter-task event flows are known, CODA can consider each one for mapping to either: 1) a software signal, 2) a tightly-coupled message, or 3) a queued message. The rule executions required for the case studies are given in Table 18. Only one of the rules, Event Flow To Message, considers design decisions that depend upon the level of a designer's experience. When an experienced designer is available, CODA asks the designer whether the sending task must wait for the receiving task to accept an event before continuing. If the designer knows this information and provides it, then CODA can make a better decision. If the designer does not know the information or if the designer is inexperienced, then CODA makes a default mapping to a queued message. For the case studies in this dissertation, CODA consults the designer only twice in 64 rule executions, or for about 3% of the control and event flows.

Table 18. Rule Executions to Allocate Control and Event Flows

Rule	C1	C2	R1	R2	R3	E1	E2	E3	E4	RT	Totals
Control Flow To Software Event											
Event Flow To Software Event			4	4	4	2	2	2		2	20
Locked-State Event Flow To Tightly-Coupled Message											
Control Flow To Tightly-Coupled Message	1	1									2
Event Flow To Message	1	1							1		3
Ride On Existing Queued Message	6	6	3	3	3						21
Ride On Existing Tightly-Coupled Message	5	5							1		11
Input Event To Queued Message	2	2	1	1	1						7
Totals	15	15	8	8	8	2	2	2	2	2	64

As indicated in Table 18, each rule, save two, executes at least once during the case studies. One unexecuted rule, Control Flow To Software Event, maps control flows, that is, triggers, enables, and disables from control tasks, to software events whenever the number of control flows to be mapped falls below a threshold defined in the target environment description. This situation never arises in the case studies. Another unexecuted rule, Locked-State Event Flow To Tightly-Coupled Message, also recognizes a situation that does not arise in the case studies.

11.2.2.2.3 Data Flows

After mapping the inter-task control and event flows to specific mechanisms, CODA maps each inter-task data flow to either a tightly-coupled message or a queued message. Table 19 enumerates the rule executions required to map data flows to messages for the case studies.

Only two of the rules listed in Table 19 consider consulting the designer. One of these rules, Stimulus To Message, handles mapping decisions when insufficient information exists to allow CODA to select a precise mapping. Here, CODA consults an experienced designer about the synchronization requirements surrounding data flows between two tasks. If the designer cannot provide any help or if no experienced designer is available, then CODA maps the questionable data flow to a queued message. CODA also consults a designer upon recognizing that a tightly-coupled message is exchanged between any task and a device input/output task. In such cases, one of the tightly-coupled messages is likely to be sent in answer to the other; however, CODA might be unable to determine the precise relationship between the two messages. If the designer cannot provide any help or if no experienced designer is available, then CODA assumes that the tightly-coupled message from the device input/output task answers the corresponding message flowing to the device input/output task. For the case studies, CODA consults the designer in 20 of the 80 rule executions, or in 25% of the cases. This represents substantial interaction between CODA and the designer.

Table 19. Rule Executions to Allocate Data Flows

Rule	C1	C2	R1	R2	R3	E1	E2	E3	E4	RT	Totals
Stimulus Rides On Existing Queued Message		2	4	4	4					1	15
Stimulus Rides On Existing Tightly-Coupled Message	2										2
Response Rides On Existing Tightly Coupled Message											
Response To Tightly-Coupled Message					1					1	2
Stimulus To Control Task Rides With Events			1	1	1						3
Stimulus To Resource Monitor Or Control Task											
Stimulus From Device Input Task									1		1
Receives Only Stimuli From Multiple Senders						6	6	6	4	2	24
Stimulus For Locked State Event						1	1	1			3
Stimulus To Message	1	1	4	4	4					5	19
Stimulus For Reverse Channel			2	2	1	1	1	1	1		9
Finding Request & Response			1	1							2
Totals	3	3	12	12	11	8	8	8	6	9	80

11.2.2.2.4 Message Priorities and Queue Interfaces

When an experienced designer is available, CODA examines the queued messages received by each task. For each task that receives queued messages from multiple sources, CODA consults with the experienced designer in an effort to learn if any of the messages require varying priorities. Considering this consultation requires a single rule, executed once for each of the ten designs generated for the case studies. No means exists to distinguish message priorities without consulting a designer. For only three of the designs, E2, E3, and RT, does a designer actually provide varying message priorities.

After any message priorities are assigned, CODA examines the queued messages flowing among the various tasks in a design and maps those messages into an appropriate queuing mechanism, depending upon the message queuing services provided by the intended target system. The rule executions required to make these decisions for the case studies are enumerated in Table 20. None of the rules listed in Table 20 requires consultation with a designer. One rule, Single Priority, Only Priority Queues Available, handles situations where queued messages are exchanged between tasks at only one priority but where the intended target system provides only priority message queues. This combination of requirements should be very rare; no such situation occurs among the case studies.

11.2.2.3 Module Structures

The structuring of modules requires a series of six decision-making processes. Each process is discussed in turn below.

11.2.2.3.1 Candidate Modules

Table 21 shows the number of rule executions required to identify candidate modules for each of the case studies. The heuristics that identify candidate modules were easily automated because previous analysis of the data/control flow diagrams produced a

Table 20. Rule Executions to Allocate Queue Interfaces

Rule	C1	C2	R1	R2	R3	E1	E2	E3	E4	RT	Totals
Single Priority, Message Queue Available	1	2	4		4	4	2	2	3		22
Single Priority, Only Priority Queues Available											
Single Priority, No Queues Available				4						2	6
Multiple Priority, Only Message Queues Available							2				2
Multiple Priority, Priority Queues Available								2			2
Multiple Priority, No Queues Available										1	1
Totals	1	2	4	4	4	4	4	4	3	3	33

Table 21. Rule Executions to Identify Candidate Modules

Rule	C1	C2	R1	R3	E1	E4	RT	Totals
Device-Interface Module	12	12	5	5	8	8	3	53
State-Transition Module	2	2	1	1	1	1		8
Data-Abstraction Module	11	11	2	2	1	1	1	29
State-Dependent Function-Driver Module	1	1	1	1				4
Triggered Algorithm-Hiding Module								
Subsystem Interface Module								
User Interface Module								
Totals	26	26	9	9	10	10	4	94

semantic interpretation of the various transformations. As shown in Table 21, three of the seven rules were not executed for the case studies. These rules recognize situations that do not arise in the case studies. None of the case studies included terminators representing user roles or external subsystems. In addition, none of the case studies required grouping triggered transformations, except where those transformations operated on a virtual device or a data store.

11.2.2.3.2 Functions of Data-Abstraction Modules

During the next decision-making process CODA attempts to allocate transformations to data-abstraction modules, identified during the previous decision-making process. Table 22 enumerates the required rule executions for the case studies. Most of the heuristics were straightforward to specify. One rule, Connects With Multiple Data Stores, handles situations where a transformation interacts with multiple data stores but where CODA cannot make a definite decision about allocating the transformation. This rule is only used when an experienced designer is available. The rule consults the designer to see if the designer can make an appropriate allocation. Where an experienced designer is unavailable or cannot indicate an appropriate allocation then the transformation is left for later decision-making processes. Where the designer does indicate an allocation, then another rule, Function Allocated To Data Store, implements the designer’s decision. In the case studies, the design generator did not need to consult a designer regarding this issue, and so these two rules were never executed.

Table 22. Rule Executions to Allocate Functions to Data-Abstraction Modules

Rule	C1	C2	R1	R3	E1	E4	RT	Totals
Connects With Only One Data Store			1	1				2
Writes To Only One Data Store	11	11	2	2	3	3	1	33
Sole Reader From Data Store	5	5			1	1	1	13
Connects With Multiple Data Stores								
Function Allocated To Data Store								
Totals	16	16	3	3	4	4	2	48

11.2.2.3.3 Remaining Transformations

During the third decision-making process CODA examines transformations not yet allocated to a module in an attempt to place each of them in an existing module, or to allocate a new module for each. The rule executions required to make these decisions for the case studies are enumerated in Table 23. The heuristic that allocates a module based on each asynchronous function in a data/control flow diagram is straightforward. In general, however, CODA could not make decisions to allocate synchronous functions to existing modules because such allocations require application-specific knowledge, unavailable to CODA. For synchronous functions, one rule, User Specifies Allocation, consults an experienced designer, where available. If an experienced designer indicates an allocation, then another rule, Synchronous Function Allocated To IHM, implements the designer’s decision. Where the designer does not make an allocation or where no experienced designer is available, then the transformation in question is allocated in the next decision-making process.

Table 23. Rule Executions to Allocate Remaining Transformations to Modules

Rule	C1	C2	R1	R3	E1	E4	RT	Totals
Active Functions			2	2	1	1	5	11
Synchronous Function Allocated To IHM			2	1			3	6
User Specifies Allocation			2	1			5	8
Totals			6	4	1	1	13	25

11.2.2.3.4 Isolated Elements

Next CODA considers for allocation any unallocated transformations or data stores remaining on a data/control flow diagram. The rule executions that consider these allocations for the case studies are listed in Table 24. One rule, Isolated Function, allocates a module for each function not allocated to a module by previous decisions. In the case study, all functions were allocated previously, and so this rule never executes.

Table 24. Rule Executions to Allocate Isolated Elements to Modules

Rule	C1	C2	R1	R3	E1	E4	RT	Totals
Isolated Update	2	2						4
Isolated Function								
Isolated Data Store			1	1				2
Totals	2	2	1	1				6

The two remaining rules in this decision-making process deal with isolated data stores. One rule, Isolated Update, recognizes situations where a data store interacts only with one transformation and where the interaction is an update. CODA understands that two possibilities exist: 1) the data store provides local storage for the updating transformation or 2) the data store provides an interface to another subsystem that is unknown to CODA. CODA asks the designer to indicate the pertinent case. CODA then takes an appropriate action with respect to the data store. Each data store that remains unallocated is mapped to a distinct data-abstraction module by the remaining rule, Isolated Data Store.

11.2.2.3.5 Module Subsumption

After allocating each transformation and data store on a data/control flow diagram to some module, CODA examines the relationships between data-abstraction modules. Where one data-abstraction module is read by only one other module and the reading module is also a data-abstraction module, the possibility exists to have the reading module subsume the module from which it reads. Whether this subsumption should occur depends upon application-specific knowledge unavailable to CODA. For this reason, when an experienced designer exists, CODA consults the designer about each such case in the evolving design. CODA takes an appropriate action as directed by the designer. Where no experienced designer exists CODA does not consider subsuming data-abstraction modules. The appropriate rule executions for the case studies are given in Table 25.

Table 25. Rule Executions to Consider Module Subsumption

Rule	C1	C2	R1	R3	E1	E4	RT	Totals
Exclusive Read Between DAMs	2							2
Totals	2							2

11.2.2.3.6 Module Operations

During the final decision-making process needed to structure modules, CODA determines the operations provided by each module and the parameters needed for each operation. These decisions could be left to the designer because a range of different mappings is possible depending upon the designer's preferences. Instead, CODA uses specific mapping rules to create operations and parameters from a data/control flow diagram and the evolving design. The rules operate without consulting the designer. Later, if desired, the designer can alter the results from these automated mappings. Table 26 displays the rule executions required to determine module operations for the case studies. Only two of the mapping rules were not executed during the case studies. Both rules involve a stimulus entering and a response leaving a single transformation, where the stimulus and response have the same name. Each rule maps the stimulus-response pair to an input/output parameter for the appropriate operation. None of the case studies included such a situation.

Table 26. Rule Executions to Determine Module Operations

Rule	C1	C2	R1	R3	E1	E4	RT	Totals
Allocate Arc Internal To IHM	19	18	5	4	4	4	5	59
Interface Module Skeleton	12	12	5	5	8	8	3	53
Signal To Interface Module					5	5		10
Signal From Interface Module	15	15	4	4	4	4		46
Stimulus Without Response From Interface Module			2	1	3	3	1	10
Stimulus Without Response To Interface Module	8	8	6	5	3	3	3	36
Stimulus With Response To Interface Module	5	5		1			1	12
Same Stimulus And Response With Interface Module								
Interface Module Without Drivers	1	1	2	2				6
STM	2	2	1	1	1	1		8
DAM Get	19	20	3	3				45
DAM Put	1	1	1	1				4
DAM Update								
External Function	19	19	11	10	5	5	9	78
External Function Invocation	12	12	7	7	1	1		40
Deactivate Module	3	3						6
Multiple Signals To Function					1	1		2
Signals From Function	1	1	4	6	3	3	1	19
Same Stimulus Response With Function								
Stimulus To Function	1	1	7	6	6	6	8	35
Stimulus Or Response From Function	9	9	10	9	6	6	7	56
Totals	127	127	68	65	50	50	38	525

11.2.2.4 Task and Module Integration

During the remaining design phase CODA integrates the task and module structures into a concurrent design. Three decision-making processes compose this phase. The rule executions for each of the three processes, applied to the case studies, are given in Tables 27, 28, and 29. All of these rules operate without consulting the designer. Two rules, Place SIM and Place UIM, are not executed for the case studies because no user roles or external subsystems appear. Two additional rules, Place Selected IHMs With Incompatible Cardinality and Operation Yields Data, also do not execute because the rules recognize situations that do not appear in the case studies. Calling sequences between tasks and modules can be arranged using a variety of techniques depending upon decisions made by a designer. Instead, several rules, listed in Tables 28 and 29, encode mapping of specific situations to specific calling sequences. In this way, the designer need not be consulted on these issues.

11.2.2.5 Assessment of Automated Design Generation

How successful was the approach taken to automate the generation of concurrent designs? For the case studies described in Appendices B through E, the CODA design generator made 1,568 design decisions. CODA made 1,524 design decisions, or about 97%, without consulting the designer. CODA consulted an experienced designer regarding only 44, or about 3%, of those design decisions. Table 30 enumerates the types of decisions where CODA asked for help. Had an experienced designer been unable to help, CODA would have taken default decisions for each of these cases.

Table 27. Rule Executions to Place Modules

Rule	C1	C2	R1	R3	E1	E4	RT	Total
Place STM	2	2	1	1	1	1		8
Place DIMs For Asynchronous Devices	2	2	3	3	3	3	3	19
Place DIMs For Periodic Devices	6	6	2	2				16
Place Selected Shared IHMs	2	2						4
Place Selected Captive IHMs	3	3	3	3	3	3	8	26
Place Selected IHMs With Incompatible Cardinality								
Place DAMs	10	11	3	3	1	1	1	30
Place DIMs For Physically Contained Passive Devices					3	3		6
Place SIM								
Place UIM								
Total	25	26	12	12	11	11	12	109

Table 28. Rule Executions to Link Tasks and Modules

Rule	C1	C2	R1	R3	E1	E4	RT	Totals
Invocation Via Transformation	15	14	2	2	4	4		41
Invocation Via Parameter Matching	1	1						2
Invocation Via Data Store	3	4	2	2				11
Totals	19	19	4	4	4	4		54

Table 29. Rule Executions to Link External Modules

Rule	C1	C2	R1	R3	E1	E4	RT	Totals
Transformation Requires Transformation	5	5						10
Transformation Requires Data Store	15	14						29
Operation Takes Data		1						1
Operation Yields Data								
Totals	20	20						40

Table 30. When CODA Asks the Designer for Help

Type of Decision	C1	C2	R1	R2	R3	E1	E2	E3	E4	RT	Totals
Mapping a Data Flow to a Message	1		5	5	4					5	20
Allocating a Synchronous Function to a Module			2		1					5	8
Allocating a Synchronous Function to a Task			2		1					3	6
Assigning Priorities to Messages								1	1	1	3
Combining Modules	2										2
Allocating an Isolated, Update-Only Data Store to a Module	2										2
Mapping a Control Flow or Event Flow to a Message	1								1		2
Combining Tasks with Compatible, but not Identical, Periods	1										1
Totals	7		9	5	6			1	2	14	44

11.3 Aspects of the Approach that Proved Difficult to Automate

Certain aspects of the analysis of data/control flow diagrams and the generation of concurrent designs proved difficult to automate. In such cases CODA consults an experienced designer, if available. Where an experienced designer is not available, or where the designer cannot provide any additional insight, CODA takes a default decision. Default decisions generally lead to an increased number of tasks and modules and to the use of queued messages where tightly-coupled messages might otherwise be appropriate. Some specific problems are discussed below.

11.3.1 Specification Analysis

For the case studies described in Appendices B through E, approximately 86% of the semantic concepts could either be represented directly with a data/control flow diagram or inferred from such a diagram using the classification rules specified in Appendix A. CODA could not classify the remaining 14% of the semantic concepts without consulting the designer. Where a designer cannot help with the classification, CODA uses default decisions to classify a concept. The difficult cases are identified below.

11.3.1.1 Aperiodic Functions

In some cases, where an aperiodic function might be classified as either asynchronous or synchronous, CODA's classification rules make a tentative classification and then CODA asks the designer to confirm or override the decision. In other cases, CODA's classification rules cannot even make a tentative classification for an aperiodic function. Where the designer cannot provide assistance, CODA classifies an aperiodic function as asynchronous.

In general, an aperiodic function might be classified as synchronous whenever the function will not unduly delay any invoking transformation and whenever that function does not take a substantial amount of time to execute. These factors are not shown on a data/control flow diagram. One simple approach to address this problem might be to include attributes in the mini-specification for each transformation. One of these attributes could specify the estimated, maximum execution time for the processing within the transformation. A more sophisticated approach to overcome the problem might be to define a language for expressing the pseudo-code within data transformations. Such a language could include information about the time taken to execute sequences of statements in the pseudo-code and about the number of iterations expected for each invocation of looping constructs within the pseudo-code. A similar approach is taken in the MARS project, conducted by researchers at the Technical University of Vienna. [Pospischil92] The MARS project allows each task to be specified using Modula R, an extension of Modula 2 that can be marked with primitives to facilitate the timing analysis of a task. No matter which approach is adopted, the information recorded might serve as useful input to an automated performance analysis of generated designs.

11.3.1.2 Triggered Functions Receiving Data Flows

Another situation arises where CODA's classification rules require additional information. When a triggered function receives a data flow from another transformation, CODA's classification rules cannot infer whether the triggered function can finish during a state transition. This problem occurs because no means exist to establish whether or not a message containing the data flow will always be present at the time the function is triggered. In such cases, CODA must consult a designer to elicit the missing information. When a designer cannot provide assistance, CODA classifies the function as a triggered, synchronous function.

One approach to overcome this problem is for the designer to avoid writing specifications that allow data flows directly to triggered functions. A less restrictive approach might be to include within a pseudo-code language for data transformations a notation for representing the reception of data flows and event flows. Such a notation could allow a designer to specify cases where an incoming data flow or event flow is expected to be available at the time the receiving data transformation is invoked. This information, coupled with the timing information gleaned from an analysis of the pseudo-code internal to the data transformation, might enable an automated classifier to infer whether a triggered function can finish during a state transition.

11.3.1.3 Stimulus versus Response

A different case that proved difficult to classify can occur when data flows are exchanged between a pair of functions. In some such cases, no evidence exists to distinguish which of the data flows, if either, is sent in response to the other. Here, a designer must be consulted. When a designer cannot help, CODA classifies neither data flow as a response.

This problem might be overcome by providing a language for specifying, within the pseudo-code for data transformations, the transmission and reception of data flows and event flows. Analysis of the pseudo-code might then facilitate automated classification of data flows as stimuli or responses, based upon the constructs specified for sending and receiving the data flows.

11.3.1.4 Classifying Terminators

Another problem relates to the classification of terminators. Any time a terminator appears on a data/control flow diagram CODA must ask a designer to indicate whether the terminator is a device, a user role, or a subsystem. This problem is ameliorated somewhat by allowing the designer to indicate with a single answer that all of the terminators in a given diagram represent devices. Whenever other types of terminators exist the designer is queried more closely about each terminator. When a designer cannot classify a terminator, CODA assumes the terminator to be a device. Adopting a naming convention for terminators could allow the designer to represent this information directly on the data/control flow diagram and, thus, to avoid a potentially tedious dialog. Alternatively, the RTSA notation could be extended to allow representation of terminator types.

11.3.1.5 Eliciting Required Information

Other situations also require consultation with a designer. Once CODA properly classifies the elements of a data/control flow diagram, some elements will require additional information that can only be provided by the designer. This involves two specific cases: 1) periods for timers and 2) maximum rates for system stimuli. CODA does not provide defaults for this information because these details are critical to the performance of the design.

Here, CODA's interactions with the designer might be reduced or eliminated by including a language for annotating arcs in a data/control flow diagram. Were such a language available, then the designer could write essential information, such as timer periods and maximum arrival rates, directly onto a data/control flow diagram.

11.3.1.6 Expressing Cardinality

In another situation, a designer might wish to specify multiple cardinality for transformations and data stores on a data/control flow diagram. In the absence of an appropriate notation, a designer must be consulted to specify any node cardinality that exceeds one. Unless otherwise instructed, CODA assumes a cardinality of one.

One approach to solve this problem is to provide attributes for annotating either nodes on data/control flow diagrams or for annotating mini-specifications. One of the attributes might allow the cardinality associated with specific nodes to be annotated. Another approach to address this problem might involve augmenting the specification with additional constructs to allow the modeling of aggregation. This is discussed further in a subsequent paragraph (see 11.3.1.7.3).

11.3.1.7 Specification Addenda

Three specification addenda, representing information that can improve the quality of design decisions, cannot be inferred from a data/control flow diagram but must, instead, be elicited from the designer. Where the designer does not provide this information, CODA assumes no specification addenda exist. Each case is identified below.

11.3.1.7.1 Mutual Exclusion

One addendum specifies exclusion groups. Two types of exclusion groups can be specified. One type contains several transformations controlled by one state-transition diagram. Future extensions to the research described in this dissertation might permit exclusion groups of this type to be established automatically by analyzing the state-transition diagrams associated with each control transformation. The second type of exclusion group contains state-independent transformations that cannot execute simultaneously. A designer might glean the information required to identify exclusion groups of this second type from reading the textual descriptions accompanying the data/control flow diagram. This information is unlikely to be inferred automatically but perhaps extensions to the data/control flow diagram notation could allow a designer to represent this knowledge directly.

11.3.1.7.2 Locked-State Events

A second specification addendum, locked-state events, cannot be inferred by CODA. Extensions to the research reported in this dissertation might enable an automated partitioning of events into two sets: 1) those that are definitely not locked-state events and 2) those that might be locked-state events. Consultation with a designer would still be required to select actual locked-state events from among the eligible set.

To fully automate this process the data/control flow diagram notation would need to be extended to express relationships among data flows and event flows entering and leaving the terminators. One such relationship might express, for example, where the input data flow from one terminator could not occur until after the output data flow to a different terminator.

11.3.1.7.3 Aggregation

A third specification addendum, aggregation groups, also cannot be represented nor inferred. CODA must ask a designer to provide this information. Extending the RTSA method to permit aggregation information to be represented directly, perhaps through an entity-relationship, or E-R, model that is mappable to components on a data/control flow diagram, could eliminate the need to elicit this information from the designer. In addition, specification of cardinalities might be tied to the data/control flow diagram through the E-R model.

11.3.2 Design Generation

For the case studies described in Appendices B through E, the CODARTS heuristics are automated to a point where CODA makes about 97% of the design decisions without consulting a designer. For the remaining design decisions, consultation with an experienced designer proved advantageous because the best decisions depended upon application-specific knowledge, unavailable to CODA. Where an experienced designer is not available or cannot provide relevant assistance, CODA takes default decisions, as necessary to generate a working, if suboptimal, design. Situations where CODA consults with a designer are identified and discussed below.

11.3.2.1 Allocating Transformations to Existing Tasks and Modules

In certain situations, a transformation proves difficult for CODA to allocate to an appropriate task or module. In particular, when a synchronous function connects, through data flows, with several other transformations that are already allocated to a task or a module, CODA alone cannot decide which of the possible allocations would be best. In addition, CODA consults a human designer during module structuring whenever a transformation cannot be clearly allocated to one among a set of data stores with which the transformation interacts. In these situations, a designer, using application-specific knowledge, might make judgments about the best allocation. When a designer cannot help, CODA allocates each transformation in question to a separate task or module.

11.3.2.2 Combining Tasks with Compatible Periods

CODA's lack of application-specific knowledge also affects, in part, decisions about whether to combine tasks with compatible, but not identical, periods. In such cases, depending upon the relative differences in importance or function of each task, a designer might choose to combine the tasks. When an experienced designer is not available for consultation, CODA chooses not to combine the tasks in question.

11.3.2.3 Combining Data-Abstraction Modules

Two situations concerning Data-Abstraction Modules, or DAMs, also cause difficulty for CODA. In one situation, a DAM might be read only by transformations within another DAM. Depending upon the functional similarity of the two DAMs, a designer might decide simply to combine them into a single DAM. CODA, lacking the required application-specific knowledge, refers such questions to an experienced designer. Where no experienced designer is available, CODA chooses not to combine the DAMs in question.

In a second situation, CODA cannot determine whether an isolated, update-only data store provides local storage for an existing module or whether the data store might hold data used by another subsystem, unknown to CODA, and should, therefore, be allocated to a separate DAM. An experienced designer, possessing the necessary application-specific knowledge, can help CODA decide how to allocate the data store in question. When an experienced designer is unavailable, CODA assumes that the data store in question provides local storage for an existing module.

11.3.2.4 Determining Synchronization Requirements

Another difficulty faced by CODA occurs whenever the synchronization requirements for data flows and event flows must be considered. To make sensible judgments in such cases, a designer applies application-specific knowledge about the processing embodied within sending transformations. CODA does not possess this knowledge. This lack of knowledge impedes automatic decision-making when mapping data flows, particularly, and event flows, to a lesser degree, to either queued or tightly-coupled messages. To compensate for this difficulty CODA encodes two convenient assumptions within the design-decision rules. First, every event flow and data flow transmitted by a message and going in the same direction between a pair of tasks is assumed to map to the same type of message. This assumption prevents having to consult the designer about the unique synchronization requirements of each event flow and data flow between tasks. A second assumption is that, in the absence of any additional information, the data flows going in opposite directions between the same pairs of tasks can be mapped to the same type of message. Even with these assumptions, whenever doubt exists, CODA must consult an experienced designer to make the initial determination of synchronization requirements for messages exchanged between a pair of tasks. When an experienced designer cannot help, CODA simply maps questionable data and event flows to queued messages.

11.4 Quality of Generated Designs

In addition to assessing the degree of automation achieved with the prototype, the quality of the generated designs should also be considered. Given that the approach automates heuristics used by a human designer to generate concurrent designs from data/control flow diagrams, designs generated by CODA can be compared against designs generated from the same input by a human designer. The four case studies chosen for this dissertation consist of real-time problems, specified with text, data/control flow diagrams, and, where applicable, state-transition diagrams, taken from the literature. For each of these problems, a design, generated by a human designer, exists in the literature. This allows CODA's solutions to be compared with existing solutions from human designers. For the case studies, the designs generated by CODA appear reasonably consistent with, though not always identical to, the preexisting designs. Each phase of the design-generation process is considered below.

11.4.1 Task Structure

For two case studies, the automobile cruise control and monitoring system and the robot controller, CODA generated a task structure identical to that provided by Gomaa. For the elevator control system, CODA could not treat the Elevator Manager and the Elevator Controller tasks differently when considering task inversion. Gomaa, however, did treat the two tasks differently, inverting the Elevator Manager but not the Elevator Controller. Apparently, this discrepancy exists because the heuristics for task inversion, as specified within CODARTS, are not given in sufficient detail.

For the remote temperature sensor application, CODA generated a task structure that included the same number of tasks as in a design provided by Nielsen and Shumate. Two differences exist, however, in the specifics of the tasks. Nielsen and Shumate define a task for relaying DP ACK signals from one task, Rx Host Message, to another, Determine Host Output. Since the target environment description used by CODA for this problem assumes that software signals can be sent directly between tasks, this relay task is not needed in the CODA solution. On the other hand, the design generated by CODA determines that messages sent from one task, Determine Host Output, to another task, Tx Host Message, should be queued. Since no message queuing is available, CODA creates a queue-control task to hold the queued messages. Nielsen and Shumate decide that the messages sent between the same two tasks should be tightly-coupled; thus, they do not provide a queue-control task for that interface. Both differences in the two designs result from auxiliary tasks defined to provide interfaces between the main tasks in the design. The basic tasks in both designs are identical; though this would not necessarily be the case if the designer provides different information to CODA during the numerous interactions required to solve this problem.

In the case of the remote temperature sensor, the resulting task structure depends to a large degree on information provided by the designer relative to the classification of aperiodic functions on the data/control flow diagram, and relative to the allocation of synchronous functions that border on two or more tasks. This information is application-specific; a quite different task structure can be generated depending upon what the designer says, if anything, regarding these questions. In general, then, CODA provides task structuring that appears very close to the human designer's solution whenever the input data/control flow diagram takes maximum advantage of the semantic model provided by COBRA. Without this, the task structure generated by CODA is highly dependent upon any information elicited from the designer.

11.4.2 Module Structure

CODA's module structuring decisions are close, but not always identical, to the solutions given in the literature. CODA's module structuring for the automated cruise control and monitoring system is identical to Gomaa's, save in one particular. CODA generates a device-interface module for the time-of-day clock. Gomaa assumes that this function is built into the operating system and, thus, that no interface module is necessary. Here, a difference in assumptions made by CODA and the human designer lead to a difference in the solution.

Similarly, CODA's module structuring solution for the robot controller matches Gomaa's solution quite closely. CODA does, however, generate three modules that Gomaa does not. Two of these modules result from CODA's insistence[1] that each transformation and data store on the data/control flow diagram be mapped to a module. Two data transformations, Interpret Program Statement and Generate Axis Command, form the basis for algorithm-hiding modules generated by CODA. Additional data transformations are placed inside these modules by CODA based on information provided by the designer. Gomaa simply does not map these data transformations to modules.

Another difference between CODA's solution and Gomaa's solution occurs because CODA creates a data-abstraction module, based upon a data store, Program ID, and a function-driver module, based upon five data transformations that send data flows to a single, virtual device, Output to Panel. Gomaa combines these two modules, allowing the function-driver module to subsume the data-abstraction module. Here, Gomaa makes a judgment beyond CODA's reach. A rule could be included within CODA to recognize situations such as this, and then to refer them to an experienced designer, but the ultimate decision to combine the modules must be made by the designer, based upon application-specific knowledge.

[1] Remember one goal of CODA is to allocate each element in the specification to at least one component in the design.

For the elevator control system, CODA's module structure is identical to Gomaa's, except that CODA provides an algorithm-hiding module for the Scheduler. Again, this is based on CODA's goal that each transformation and data store be assigned to a module.

In the case of the remote temperature sensor, comparison between the solution provided by Nielsen and Shumate and CODA's solution proves difficult because Nieslen and Shumate do not map the data flow diagram to modules, instead they directly define Ada packages based upon their task structure. In general though, CODA's module structuring for the remote temperature sensor is driven in large part by information elicited from the designer. A range of module structures is possible, depending upon the advice given by the designer. This indicates that for module structures to be generated most automatically by CODA, the data/control flow diagram should take maximum advantage of COBRA's semantic model. In such cases, the module structures generated by COBRA appear reasonably close to those generated by a human designer. Where data/control flow diagrams are constructed without adhering to COBRA's semantic model, the resulting module structure depends greatly upon information elicited from the designer.

In addition to structuring modules, CODA goes on to determine module operations, including parameters. In general, the creation of module operations is highly idiosyncratic, depending upon a range of considerations too detailed to represent easily in design rules. Since CODA pursues a goal of allocating each element in the specification to an element in the design, rules are defined to map nodes and arcs to module operations.

In most cases, the operations generated by CODA appear to be reasonable, though not always matching the operations provided in the solutions given in the literature. In one case, the elevator control system, CODA generated an operation for a data-abstraction module that had been overlooked by Gomaa. Less reasonable is CODA's mapping of specification elements to operation parameters. Here, simple mapping rules are used to ensure all specification elements are mapped to design elements. In many cases, a human designer might choose to change the specification of operation parameters generated by CODA.

11.4.3 Task and Module Integration

To integrate the task and module views within a concurrent design, CODA must determine which modules are accessed by a single task and which modules are shared by multiple tasks. After making this determination, CODA must go on to identify the calling relationships among tasks and shared modules. These decisions, as made by CODA, appear consistent, for the most part, with those made by the human designers. A few differences are worth noting.

In the automobile cruise control and monitoring case study, CODA identifies each of two device-interface modules, Mileage Display and Maintenance Display, as being accessed solely by a single task, Compute Average Mileage and Check Maintenance Need, respectively. This leads CODA to place each of these modules within the accessing task. Gomaa chooses, instead, to keep these modules outside any task. Gomaa's choice reflects a design structure where each module that generates information to be displayed calls directly to the display modules in order to write the information. Since the calling modules are placed outside any task, the called modules are also placed outside any task. In general, to handle complex relationships between tasks and multiple, shared modules, a designer can choose several approaches. For example, when a task must invoke two operations, in separate modules in sequence, and the second operation requires information from the first, three possibilities exist: 1) the task calls the operation in the first module and that operation calls the operation in the second, passing any needed data as an input parameter to the second operation; 2) the task calls the operation in the first module and that operation returns data as an output parameter that the task then passes as an input parameter when calling the operation in the second module; 3) the task calls the operation in the first module and then the operation in the second module, and the second operation calls another operation in the first module to get any data needed. To simplify decision-making and limit interaction with the designer, and also to avoid unnecessary coupling, CODA consistently uses the second mapping; thus, when a human designer chooses one of the other approaches, as Gomaa does for example in his design for the automobile cruise control subsystem, CODA's design differs.

11.4.4 Task Interfaces

In most instances, the task interfaces generated by CODA for the case studies agree exactly with those provided by the preexisting solutions. Differences that do, or might, exist result from one of three factors. Sometimes, CODA adopts a consistent mapping rule in cases where a human designer might decide to devise a different mapping. For example, in the elevator control system, CODA maps two event flows, Up Request and Down Request from the Elevator Manager task to the Elevator Controller task, onto software signals. This mapping is made because the target environment used in the case study allows up to two software signals between each pair of tasks. Gomaa chooses an alternative mapping. This difference illustrates that in some situations the rules defined for CODA lead consistently to specific mappings, even though a designer might choose to use more judgment when applying the comparable CODARTS heuristic.

CODA might also generate task interfaces that differ from a preexisting solution in cases where CODA relies on a designer’s judgment. No such differences appear within the case studies because the designer interacting with CODA always provides information leading CODA to make decisions consistent with the decisions taken by the human designer who developed the preexisting solution. This need not have been the case. For example, in the robot controller application, Motion Blocks and Motion Ack Queues are exchanged between two tasks using queued messages. Queued messages were selected by CODA because the designer indicated that inter-task synchronization was not required for these data flows. Had the designer indicated that synchronization was necessary then CODA would map the same data flows to tightly-coupled messages.

On the other hand, where CODA can benefit from a designer’s advice but where no experienced designer is available, CODA might also generate task interfaces that differ from a preexisting solution. In effect, the default mapping decisions that CODA takes, in the absence of any advice, can cause CODA to choose queued messages in situations where a human designer might choose tightly-coupled messages. Exactly this situation arises in the case of the automobile cruise control and monitoring system, when CODA generates a design for a novice designer. The interface between the Auto Speed Control task and the Throttle task, mapped by Gomaa to a tightly-coupled message, is mapped instead by CODA to a queued message.

11.5 Interpretation of Results

Given the foregoing evaluation, what can one conclude about the scope of applicability of the approach proposed in this dissertation for automating the generation of concurrent designs? First, the effectiveness of the approach varies depending upon the manner in which the RTSA notation is used to model the problem behavior. When the designer models the problem using a data/control flow diagram that makes maximum use of the COBRA guidelines and semantics, as for example in the automobile cruise control system and elevator control system case studies (see Appendices B and D, respectively), then the approach embodied in CODA proves most effective, and requires the least amount of interaction with the designer. The key to success is modeling the problem based mainly upon objects (device interface objects, control objects, algorithm objects) and data stores, while limiting the use of functions to provide only operations on data stores. When the problem is modeled using the guidance of RTSA but without the guidelines and semantics of COBRA, as for example in the robot controller case study (see Appendix C), then CODA still yields reasonable performance, although the amount of interaction with the designer increases somewhat. Using RTSA guidelines only, CODA can still view many of the edge transformations and control transformations as objects, and can detect the presence of certain algorithm objects. In such cases, CODA can also map functions that interact with data stores onto operations on those data stores. The increased interaction between CODA and the designer results from the presence of functions that do not interact with data stores. Such functions begin to transform the data/control flow diagram into a functional decomposition. In fact, CODA does not perform very well for problem models that are based completely on functional decomposition. This limitation shows up in the remote temperature system case study (see Appendix E). In the case of the remote temperature system, CODA requires substantial interaction with a designer to elicit application-specific information and hints about the degree of functional cohesion exhibited among the functions depicted on the data flow diagram. As shown in Appendix E, when CODA is asked to generate a design for a data flow diagram created using functional decomposition, then the resulting design depends to a great degree upon guidance proved by the designer. When no designer guidance is available, CODA generates a design that mirrors closely the data flow diagram.

A second issue to consider is the scalability of the proposed approach. Given that a problem model takes maximum advantage of the COBRA guidelines and semantics, how large of a problem can be tackled by the proposed approach to automated design generation? The proposed approach imposes no limits on scalability that are not already present in RTSA and COBRA. Since RTSA notation provides for a hierarchical decomposition of a problem model, fairly large problems can be represented and managed intellectually as subproblems. However, since CODA processes a data/control flow diagram as a flattened hierarchy, the effectiveness of CODA can be enhanced by introducing a graphical user interface that connects CODA's reasoning processes to the user directly through pictures of both the data/control flow diagram hierarchy and the evolving design. While such an interface is not part of the research reported in this dissertation, there appears to be no fundamental difficulty in providing such an interface. One scalability issue does warrant noting however. Checking a data/control flow diagram to ensure that all axioms are satisfied can take quite some time as the size of the diagram increases. For the largest diagram among the case studies, the automobile cruise control system, checking axioms takes several minutes on a 33 MHz Intel 486 processor.

A third applicability issue, related to scalability, is the potential for the approach embodied in CODA to address distributed systems. CODA addresses the generation of concurrent designs for systems projected to run on a single node, be it a single or multiple processor node. To generate designs for distributed systems, the problem must be partitioned into a set of subsystems that can each be processed on a single node. CODA can be used to generate a design for each of these subsystems. CODA does provide a subsystem interface object that can be used to represent connections between distributed subsystems; but knitting these subsystems together is left to the designer.